Lecture Notes

Lectures 1 & 2
CS389: Machine Learning, Spring 2023
University of Massachusetts Amherst

Phat Nguyen, John Raisbeck, & Cooper Sigrist

February 6, 2023
Last Updated: February 14, 2023

1 Supervised Learning (Lecture 1)

Definition 1.1 (Machine Learning)**.** A program, p, is able to perform a task, t, better according to some metric, m, when given experience, e.

Definition 1.2 (Model)**.** A *model* is some structure that uses parameters to define a function. An example of a model is $y = wx + b$, which is a linear model. In another more explicit notation, we might write a linear model as (refer to subsection 1.1 for more info):

$$\hat{y} = \sum_{j=1}^{m} w_j \cdot x_j + b = X \cdot W + b, \tag{1.1}$$

where we have a weight w_j for each feature x_j, a bias term b, and an output vector $\hat{y}$.

The intuition is that, the parameters (i.e. W and b) can be changed based off the data so that the model performs the optimal function/prediction/output. We say that the model learns because it is able learn the right combination of W and b to give an optimal output (global optimality is not guaranteed; we will talk more about this in the next lecture). The weights, w_j, determines how important a feature x_j is to the $\hat{y}$. When the parameters of a machine learning model are fixed, the model can be viewed as a function, F, of input features X:

$$\hat{y} = F(x_1, ..., x_m) = w_1 \cdot x_1 + ... + w_m \cdot x_m + b. \tag{1.2}$$

For example, the inputs of a weather model can be the humidity, temperature, air speed, etc., and the output could be *how much* it will rain (a type of model known as a *regression model*) or a probability distribution of *what* the weather will be like (a type of model known as a *classification model*). In a regression model the output $y \in \mathbb{R}$ while the output of a classification model is a probability distribution $Y \sim p(x)$. A classification model with 2 classes is a binary classification model, and a model with 2 or more classes is a multi-class classification model.

Definition 1.3 (Supervised Learning)**.** A *supervised learning* system is a system in which we can train a model with labeled data. For example, given an image $x^{(1)}$ the label $y^{(1)}$ could be "dog". A collection of these is a *dataset*, D:

$$D = \left\{ (x^{(1)}, y^{(1)}), ..., (x^{(n)}, y^{(n)}) \right\}, \tag{1.3}$$

where $x^{(i)}$ is the input vector to the i^{th} sample and $y^{(i)}$ is the corresponding label to the i^{th} sample[1]. The goal in supervised learning is to learn a function F such that for a new pair of data (x, y), we have $F(x) \approx y$ with high probability [3].

1.1 Why $\hat{y} = X \cdot W + b$?

Resource: If you think you would benefit from a refresher on matrix multiplication, consider this Khan Academy lesson.

Naively, we could implement a linear model using Algorithm 1.

Algorithm 1 Naive Linear Model

```
ŷ = b
for j in range(m) do
    ŷ += w_j · x_j
end for
```

However, this is a very ugly implementation and for-loops in Python are slow. For our implementation, it would be better to express the operation using matrices. Generally we want X to have shape $(n \times m)$ where n is the number of data and m is the number of features (i.e each row/data has m features), therefore a $(m \times n)$ shape would not make sense. We also want the output to have shape $(n \times p)$ where p are the predictions (i.e. each row/data has p predictions). The problem with $W \cdot X$ is that we need to transpose the X matrix which does not make sense to do so:

$$(?, ?) \times (n, m) \to (n, p). \tag{1.4}$$

Instead, writing it as $X \cdot W$ makes more sense:

$$(n, m) \times (?, ?) \to (n, p), \tag{1.5}$$

where the weight matrix can now have shape (m, p). However, we want to represent the weight matrix as shape (p, m) as it is equivalent to saying "a prediction p can be made if given m features". Therefore, the dimensions of X, W, and output will be:

- $W \to (p, m)$
- $X \to (n, m)$
- Out $\to (n, p)$

This is the convention used by Pytroch (torch.nn.Linear)[2]. Later in the course, we will see that representing W and X as matrices in form is easier to differentiate for gradient descent.

2 Regression (lecture 1)

Definition 2.1 (Linear Regression)**.** If we have a linear model (i.e. $y = X \cdot W + b$) finding the particular parameters (W and b) for this linear function is called *linear regression*. You may recall procedures for finding the line of best fit from a previous course—this is the same process, in an arbitrary dimension.

[1] The reason why we use the $X^{(i)}$ notation is because $X^{(i)}$ is a vector and sometimes we need to refer to the elements of that vector. We also want to use $x^{(i)}$ so that the convention is consistent

[2] https://pytorch.org/docs/master/generated/torch.nn.Linear.html

3 Loss Function (lecture 2)

Definition 3.1 (Loss Function)**.** A *loss function* measures how *poorly* a model performs. The lower the loss, the better. A loss of zero means it makes perfect predictions. An example of a well-known loss function is the *Mean Absolute Error* (MAE):

$$\text{MAE}(y, \hat{y}) = \frac{1}{N} \sum_{n}^{N} y^{(n)} - \hat{y}^{(n)}, \tag{3.1}$$

where $y^{(n)}$ is the n^{th} label in the data and $\hat{y}^{(n)}$ is the output of the model when given the n^{th} input, $x^{(n)}$. Another loss function is the *Mean Square Error* (MSE) and can be written formally as:

$$\text{MSE}(y, \hat{y}) = \frac{1}{N} \sum_{n}^{N} (y^{(n)} - \hat{y}^{(n)})^2. \tag{3.2}$$

In general, the line of best fit for the mean absolute error tends to produce models with more outliers than the line of best fit for the mean squared error. It is common practice to normalize the loss by the total number of training samples, N, so that the output can be interpreted as the average loss per sample [3]. The loss function gives us a quantitative way of describing the accuracy of our prediction, as visualized in Figure 1.

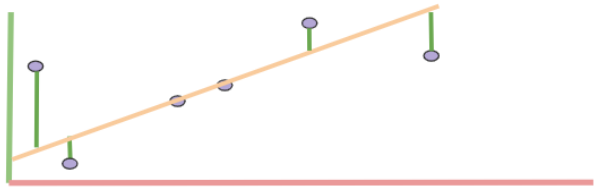


Figure 1: Loss function

Remark 3.1 (Properties of a Loss Function)**.** When configuring a machine learning system to solve a problem, one of the most important issues is determining what loss function to use. Here are some important properties for the success of a loss function.

1. The minimum of the loss function should correspond to the line of best fit,
2. The loss should be defined for all outputs,label pairs
3. The loss function should be differentiable (for gradient descent).

Therefore, an ideal loss function, $L(W)$, is convex, continuous, and differentiable.

4 Gradient Descent (lecture 2)

Resource: This topic is covered well by many authors, so we won't restrict you to our treatment. In particular, you may want to check out 3Blue1Brown's video, which Cooper mentioned in class.

In the previous section, we described two common loss functions that describe how poorly the model performs on a task. Our goal is to find the optimal set of parameters—that is, the parameters which minimize the loss:

$$W^* = \arg\min_{W} L(W) \tag{4.1}$$

Clearly, this is an optimization task. The most common tool for finding the optimum in a problem such as this is known as *gradient descent*. There are other ways of performing optimization (e.g. LBFGS) but gradient descent is the most common and established way of optimizing Neural Network loss functions [4].

Definition 4.1 (Gradient of Loss)**.** The *gradient of loss*, ∇Loss, is the *direction of steepest ascent*. The negative gradient of loss, $-\nabla$Loss, is the direction of steepest descent. The gradient of loss with respect to the weight vector can be written as:

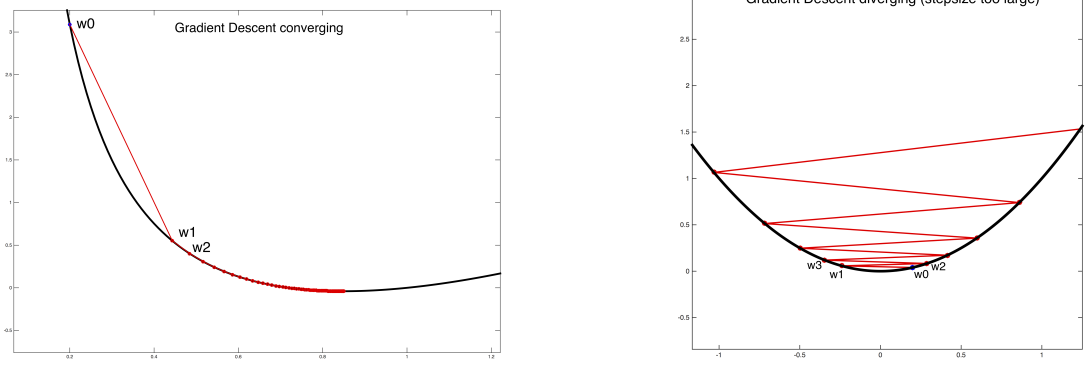
$$\nabla_W \text{Loss} = \frac{\partial L}{\partial W} = \left(\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \ldots, \frac{\partial L}{\partial w_m} \right) \tag{4.2}$$

Definition 4.2 (Gradient Descent)**.** Gradient descent is an optimization algorithm that minimizes the loss $L(W)$. The intuition behind the *gradient descent* algorithm is to iteratively update w_i in the direction of the negative gradient (the direction of steepest descent) until we converge to the weights w_i which minimize the loss function $L(w_i)$, which can be denoted as w_i^*. By multiplying the negative gradient by a sufficiently small number α, we can take a step guaranteed to reduce loss. When these steps are taken repeatedly, it is guaranteed that the process will converge to a minimum. The number α is known as the *learning rate*, and is considered a *hyperparameter*; a parameter which is a part of the learning algorithm, but not a part of the model itself. Then, the gradient descent step is:

$$W \leftarrow W - \alpha \frac{\partial L}{\partial W} \tag{4.3}$$

$$W_i \leftarrow W_i - \alpha \frac{\partial L}{\partial w_i} \tag{4.4}$$

Whether a learning algorithm converges and whether it converges to an optimal solution can depend upon the learning rate; if α is too large, as depicted in Figure 2 (b), taking steps in the direction of the negative gradient which are too large can result in moving further and further away from the optimum. A learning algorithm is said to have converged to a solution if the gradient of loss at the current parameters is 0.



(a) α is just right

(b) α is too large

Figure 2: Consequence of choosing α [3]

4.1 Batch, Mini Batch, and Stochastic Gradient Descent

Definition 4.3 (Epoch)**.** An epoch is one complete pass of the training dataset

Definition 4.4 (Batch Gradient Descent)**.** *Batch Gradient Descent*, sometimes known simply as gradient descent, describes a method that updates the parameters (W and b) using the *entire training dataset* in each step of training. For example, one would use the mean of the gradients of all the training data to update the parameters.

Definition 4.5 (Stochastic Gradient Descent)**.** *Stochastic Gradient Descent*, also known as on-line gradient descent, describes a method that updates the parameters after each piece of training data. Usually, when SGD is mentioned people are not referring to this version, but to the more common *minibatch* gradient descent. The stochastic gradient descent is technically a minibatch gradient descent with a batch size of 1.

Definition 4.6 (Minibatch Gradient Descent)**.** *Minibatch Gradient Descent* is a method that updates the parameters on every subset with a batch size, B, of the training dataset per learning step. The batch size is a hyperparameter which is usually chosen in view of memory constraints.

References

[1] Cooper.

[2] Mengye Ren and Matthew MacKay, UoT CSC 411, Winter 2019. https://web.archive.org/web/20190513201539/https://www.cs.toronto.edu/~mren/teach/csc411_19s/lec/lec06.pdf

[3] Kilian Q. Weinberger, Cornell CS4780, Fall 2018. https://web.archive.org/web/20180827054501/https://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote01_MLsetup.html https://web.archive.org/web/20181016051558/https://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote07.html

[4] Fei-Fei Li, Jiajun Wu, and Ruohan Gao, Stanford CS231n, Spring 2022. https://web.archive.org/web/20230109135558/https://cs231n.github.io/optimization-1/